The Dining Philosophers Problem Revisited

Jürg Gutknecht

ETH Zürich
gutknecht@inf.ethz.ch

Abstract. We present an alternative solution to the Dining Philosophers problem that is based on Peterson's mutual exclusion algorithm for N processes, with the benefit of not using any ingredients beyond atomic read and write operations. We proceed in two steps towards a comprehensible, symmetric, and starvation-free algorithm that does neither rely on atomic test-and-set instructions nor on synchronization constructs such as monitors, signals, semaphores, locks, etc.

1 Introduction

Ever since E. W. Dijkstra posed the story of the dining philosophers as an exercise in concurrent programming in the early 1970s [1], this problem has attracted and challenged both theoreticians and programmers, and a variety of different solutions have been developed, most of them using some kind of synchronization mechanism (typically a semaphore) to control accesses to chopsticks by hungry philosophers, see for example [2]. Amazingly, although this problem is unmistakably a restricted mutual exclusion problem, we could not find any solution that makes direct use of a classical mutual exclusion algorithm. Therefore, we took the bait and tried to reuse Peterson's simple but ingenious solution to mutual exclusion published in 1981 [3].

2 Peterson's Filter Algorithm

Peterson's algorithm guarantees mutual exclusion among a fixed number of *N* processes with respect to their critical section, without making use of any synchronization constructs. The state of each process *0,…, N – 1* is captured by an array structured variable named *claiming*. For *i* fixed, *claiming[i]* serves as an "escalator" for process *i* to travel from "floor" *0* (non-critical section) to "floor" *N* (entrance to the critical section). On each floor, the shared variable *mark* is used by a newly arriving process to leave a "footprint". Using a notional Pascal-like syntax, our version of the Peterson algorithm for *N* processes looks like this:

Program 1. Peterson's mutual exclusion algorithm for N processes

```
(* state space *)
var claiming, mark: array N of integer;

(* initialization *)
var i: integer;
```

```
begin
  for i := 0 to N-1 do claiming[i] := 0 end
end

(* process nr. i *)
var k: integer;
begin
  loop
    … (* non-critical section *)
    (* entry protocol to critical section *)
    for k := 1 to N-1 do
      claiming[i] := k; mark[k] := i;
      while (exists j: j # i:
        (claiming[j] >= k) & (mark[k] = i)) do
      end
    end;
    claiming[i] := N;
    … (* critical section *)
    claiming[i] := 0 (* relinquish exclusivity *)
  end
end
```

This algorithm is also called the *filter* algorithm, see [4]. The reason is that for each floor *i* from *1* to *N - 1* the last process arriving at this floor (the one that left the last footprint", that is, the one that set *mark[k]* to *i* most recently) is a "victim" that cannot proceed. As a consequence, at most $N - i + 1$ processes can simultaneously be on floor *i* and, as a corollary, at most one process can be on floor *N* at any time, so that mutual exclusion is guaranteed. As a fine point note that the statement *claiming[i] := N* can be omitted without loss.

It is shown in [3] that the algorithm is free from *starvation* (and *deadlock*), under the obvious assumption that each process is always guaranteed to get a chance to proceed after some finite amount of time. However, note that the algorithm does not guarantee first-in-first-out handling because one process within the entry protocol can easily pass another.

3 Peterson Modified for the Dining Philosophers

Let us first recall E. W. Dijkstra's invention of the Dining Philosophers that is illustrated in Figure 1. The original formulation of the problem was this: "Five philosophers sit around a circular table. Each philosopher is alternately thinking and eating. In the centre of the table is a large plate of noodles. A philosopher needs two chopsticks to eat a helping of noodles. Unfortunately, only five chopsticks are available. One chopstick is placed between each pair of philosophers, and each agrees only to use the two chopsticks on their immediate right and left side".

Because each adjacent pair of philosophers is forced to share one chopstick but requires two of them in order to eat, appropriate synchronization of the philosophers' access to them is necessary. Therefore, at a fundamental level, we have a restricted mutual exclusion problem, and so we now try to adapt Peterson's filter algorithm to solve it.

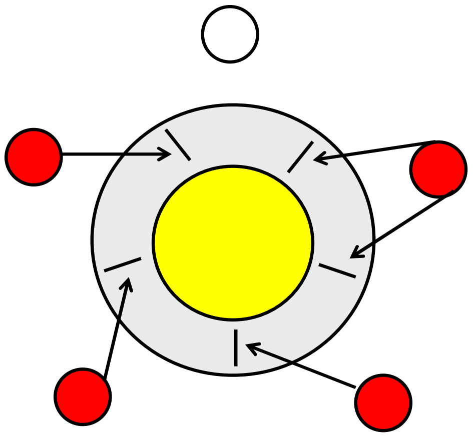

Fig. 1. A possible scenario of the Dining Philosophers: one philosopher is eating, three philosophers are waiting for their second chopstick, and one philosopher is thinking

Obviously, the *thinking* phase and *eating* phase of a philosopher's process correspond to the *non-critical* and the *critical* section of a process in the abstract setting above. The key idea of how to apply Peterson's algorithm to the philosophers problem is now straightforward: reinterpret the permission of entrance into the critical section as a mere chance to enter, and have the applicant restart his entry protocol in the case when at least one of his two neighbors is critically engaged (that is *eating*).

Keeping in mind that *(i – 1) (mod 5)* and *(i + 1) mod* 5 are the numbers of philosopher *i*'s neighbors (due to a linear array being used to represent the circular table) and using abbreviations *c* for *claiming* and *m* for *mark*, we deduce the following attempt to solve the Dining Philosophers problem for five diners:

Program 2. Attempt of a Peterson based Dining Philosopher solution

```
(* state space *)
var c, m: array 5 of integer;

(* initialization *)
var i: integer;
begin
  for i := 0 to 4 do c[i] := 0 end
end

(* activity of philosopher nr. i *)
var k: integer;
begin
  loop
    (* think *)
    loop (* claim access to chopsticks *)
      for k := 1 to 4 do
        c[i] := k; m[k] := i;
        while ((c[(i + 1) mod 5] >= k)
             | (c[(i + 2) mod 5] >= k)
             | (c[(i + 3) mod 5] >= k)
             | (c[(i + 4) mod 5] >= k))
            & (m[k] = i) do
        end
      end;
      c[i] := 5;
      if c[(i - 1) mod 5] >= 0 & c[(i + 1) mod 5] >= 0
        then exit
      end
    end;
    c[i] := -1;
    (* eat *)
    c[i] := 0
  end
end
```

According to this algorithm, each philosopher *i* is continuously cycling through the states *c[i] = 0, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, …, 1, 2, 3, 4, 5, -1, 0, 1, 2, …* whose semantics are described in Table 1.

Table 1. State diagram for philosopher processes

State c	Semantics
0	thinking
1	hungry, starting entry protocol
2	progressing in entry protocol
3	progressing in entry protocol
4	progressing in entry protocol
5	chance to eat if no neighbor eats
-1	eating

From our above discussion of the filter algorithm we know that the statements in state *c[i] = 5* (in bold type face) run under mutual exclusion, which means that *c[i] < 0* invariantly implies *c[(i - 1) mod 5] >= 0* and *c[(i + 1) mod 5] >= 0* or, in other words, that, whenever philosopher *i* is in his critical section, none of his two neighbors *(i - 1) mod 5* and *(i + 1) mod 5* are in their critical section.

However, the above solution is not free from potential starvation, as the following scenario demonstrates: an applicant *P* detects that one of his neighbors, say *Q*, is busy in his critical section and therefore immediately restarts the entry protocol. At roughly the same time, *Q* exits the critical section and, because he is still hungry, immediately requests entrance to the critical section again. This leads to a race between *P* and *Q* that might be won by *Q* because, as we know, the filter algorithm does not prevent one algorithm from passing another. Because this situation may recur any arbitrary number of times, *P* may finally starve due to the “race hazard”.

Therefore, the algorithm needs further refinement. One way to remedy the race hazard is adding a variant of the “bakery algorithm” by introducing a ticket-numbering system that (roughly) indicates the order of the processes starting the entry protocol. However, ticket-numbering may have a transitive (global) effect that prevents an otherwise unblocked philosopher from starting to eat merely because of his high ticket-number. A better refinement of Program 2 is based on a mechanism that allows a hungry but blocked philosopher in the state *c[i] = 5* to raise a flag. For this purpose, Boolean arrays *l* and *r* are added with the following semantics:

- *l[i]* ⇔ philosopher *i* would be allowed to eat but is blocked by his left neighbor
- *r[i]* ⇔ philosopher *i* would be allowed to eat but is blocked by his right neighbor

An additional guard at the end of the filter loop is now used to request each philosopher to yield to any of his neighbors who was previously blocked. The following argument shows that the resulting algorithm is free from starvation: assume that some philosopher process *i* cannot proceed from state *5* to the critical section due to one or both neighbors who are in their critical section. Then, after some finite amount of time, these neighbors will leave their critical section and will not be able to enter again before philosopher *i* has removed his flags *l[i]* and *r[i]* and has passed the critical section himself.

Program 3. The Final Program Solving the Dining Philosophers Program

```
(* state space *)
var l, r: array 5 of Boolean;
  c, m: array 5 of integer;

(* initialization *)
var i: integer;
begin
  for i := 0 to 4 do
    l[i] := false; r[i] := false; c[i] := 0
  end
end

(* activity of philosopher nr. i *)
var k: integer;
begin
  loop
    (* think *)
    loop (* enter room and claim access *)
      for k := 1 to 4 do
        c[i] := k; m[k] := i;
        while ((c[(i + 1) mod 5] >= k)
             | (c[(i + 2) mod 5] >= k)
             | (c[(i + 3) mod 5] >= k)
             | (c[(i + 4) mod 5] >= k))
             & (m[k] = i) do
        end
      end;
      c[i] := 5;
      if c[(i - 1) mod 5] < 0 then l[i] := true
      elsif c[(i + 1) mod 5] < 0) then r[i] := true
      elsif ~r[(i - 1) mod 5] & ~l[(i + 1) mod 5])
      then exit
      end
    end;
    c[i] := -1; l[i] := false; r[i] := false;
    (* eat *)
    c[i] := 0
  end
end
```

3 Conclusion

We have demonstrated the approach of adapting a well-proved generic mutual exclusion algorithm to a restricted mutual exclusion problem, with the benefit of automatically inheriting its correctness and other qualities. This approach contrasts with the usual approach of handcrafting an algorithm that solves a singular concurrency problem but inherently carries the dangers of errors due to overlooked scenarios. The net result is an elegant, symmetric, and starvation-free solution to the Dining Philosophers' problem that does neither rely on synchronization constructs nor on hardware support for atomic memory updates.

Acknowledgement

I gratefully acknowledge the hint from Daniel Kröning on Nir Shavit's proof of Peterson's filter algorithm. Also, I thank Ulrike Glavitsch, Gerardo Tauriello and Svend Knudsen for their critical inspection of the algorithm, and Brian Kirk for his constructive review of earlier versions of this paper.

References

1. Dijkstra, E. W.: Hierarchical Ordering of Sequential Processes, Acts Informatica I, 115 – 138 (1971)
2. Silberschatz, A., Peterson, J. L.: Operating Systems Concepts, Addison-Wesley (1988)
3. Peterson, G. L.: Myths About the Mutual Exclusion Problem, IPL 12(3), 115 – 116 (1981)
4. Shavit, N.: Lecture Notes for Lecture 2, Chapter 2.4.1., Tel-Aviv University, http://www.cs.tau.ac.il/~shanir/multiprocessor-synch-2003/